Simulating Shared-Memory Parallel Computers

Seth Abraham Allan Gottlieb Clyde Kruskal[1]

Courant Institute, N.Y.U.
251 Mercer Street
New York, N.Y. 10012
GOTTLIEB@NYU
{floyd,harpo}!cmcl2!gottlieb

Ultracomputer Note #70

April 1984

ABSTRACT

The design and construction of parallel computers is expensive and time consuming. Instruction level simulators represent one class of tools that can be used to ease this process. When compared to small scale prototypes, instruction level simulators offer the potential for superior instrumentation. This paper presents a simulation technique for shared-memory parallel computers, and examines the use of this technique for two very different shared-memory parallel machines.

1. Introduction

Many research groups have, in the last few years, produced high-level designs for advanced parallel computers. Since actual construction of these machines is generally very expensive and requires a great deal of time, it is important to use simulation to permit researchers the option of examining several different implementations and evaluating their potential within a reasonable amount of time and at a reasonable cost. Even after hardware prototypes are functioning reliably, simulators are useful since they can be easily modified. In this paper, we describe a simulation technique that is easy to implement and modify, executes efficiently, and is applicable to a large class of parallel computers.

[1]Current address for Seth Abraham and Clyde Kruskal is the Dept. of Computer Science, University of Illinois at Urbana. This work was supported in part by the National Science Foundation under Grants No. NSF MCS79-07804, and NSF MCS83-00981, the US Department of Energy under Contract No. US DOE DE AC02-76ER03077, and US DOE DE AC02-81ER10822, and in part by a grant from IBM.

The ideas for this simulator originated with Soapsuds [2], a simulator for an abstract parallel machine. Starting from scratch, the second author wrote the Washcloth simulator [4] of the NYU Ultracomputer [5]. The simulator was, and still is, heavily used to write and test software, and gather statistics for that machine. This information has been invaluable in making design decisions. An additional benefit is that we have gained actual experience in parallel programming, furthering insight and understanding of the difficulty involved in this endevour. Although Washcloth was specifically written to simulate the NYU Ultracomputer, a Von Neumann parallel computer, we claim it can easily be converted to simulate many shared-memory designs. For example, with some relatively minor modifications, it is currently being converted to simulate Cedar [3], a high-level data flow machine.

Section 2 very briefly describes shared-memory computers. Section 3 describes the Washcloth simulator. Section 4 reviews the NYU Ultracomputer and describes how the simulator works for it. Section 5 reviews the UI Cedar machine and describes how the simulator works for it. Section 6 summerizes the results.

2. Characteristics of shared-memory processors

We will consider shared memory MIMD parallel computers, consisting of asynchronous processing elements (PEs) each containing its own register set, program counter, and local memory. Also, each PE can access a common global memory via an interconnection network. The memory local to each PE may be a cache of this global memory, or may be separately addressed. Indeed, Cedar contains both classes of local memory. In summary, each PE has its own instruction stream, and can access both local data from its own memory, and shared data from the global memory.

3. The Washcloth simulator

The current implementation of Washcloth runs on a CDC 6600 or Cyber 70 series computer and is reasonably efficient: simulated programs suffer a slowdown factor of about 50. Each simulated PE is a Cyber augmented with whatever special instructions are needed to accomplish synchronization. These special instructions are implemented as unused Cyber machine instructions. Global memory and separately addressed local memory are provided -- a cache is not.

The Washcloth simulator exists in the form of an assembly language subroutine that can be loaded along with the routine to be simulated. A short main program calls Washcloth and passes it several parameters. One of these parameters is the start address of the code to be simulated. This code is simulated by all PEs. However, this does not mean that all PEs must all do the same thing at the same time. One of the special instructions is "read PE number". This allows PEs to be distinguished and thereby assigned different work. By convention, let PE# be a local variable that contains the identity of a specific PE, and let #PE be the total number of PEs available. Several PEs may then be assigned the same code parameterized by the variable PE#. For example, the following code distributes the *N* iterations of a loop to #PE processing elements:

```
Loop for i := PE# to N by #PE
        .
        .
        .
Endloop
```

This loop causes the first PE to execute multiple iterations with *i* successively equal to 1, #PE+1, 2*#PE+1, ... Similarly, the second PE will (simultaneously) execute iterations 2, #PE+2, 2*#PE+2, ..., and so on. Simple subroutines can be written to allow the programmer to assign loop iterations in other orders or to smaller sets of PEs.

Another parameter specified to Washcloth delineates the global memory area. All PEs can access the data in this area and must themselves perform explicit synchronization when accessing such data. Any data not in the global data area is considered to be local. Washcloth provides management of all local data in a manner totally transparent to the simulated code. Each PE has access to the same set of local variables. Whenever local variable A has a different value for any two PEs, Washcloth will automatically create an array A[1..#PE] and convert all references from A to A[PE#]. Thus storage is conserved as data variables are not expanded unless it proves necessary.

Washcloth cycles through the simulated processors executing one instruction at a time. The basic action of the simulator can best be illustrated as:

```
For PE# := #PE downto 1 do:
   Simulate one instruction for processor PE#.
```

Furthermore, a simple mechanism is provided that allows a group of instructions to be executed indivisibly on a single simulated processor. This aids in writing "read-modify-write" type operations.

User supplied "hook" routines may be called to provide the user a simple method to gather statistics. These routines are invoked at the beginning of each instruction and at each memory reference. A post processor can then calculate performance statistics from the data generated. Typical statistics generated include the total running time, number of instruction fetches, number of local memory loads and stores, number of global memory load and stores, and the number of special instructions executed.

Washcloth has been extended to simulate the action of the memory interconnection network. A precise, high-level network simulator has been implemented, but is very costly to use. Another network simulator, NETSIM, described fully in [12], dynamically approximates the delay caused by a buffered interconnection network. Also, Washcloth has been used to provide statistics relevant to caching of data memories [9].

An important feature of the Washcloth simulator is that users can write programs in a high-level language. A standard compiler is used to produce machine instructions. The special synchronization instructions are typically inserted into the high-level code by calling assembly language subroutines contained in the Washcloth library. Up till now, simulated programs have been written in Fortran and Pascal.

A disadvantage of simulating an augmented Cyber set instruction set rather than the actual instruction set being designed is that the memory access patterns and other measured quantities may be affected. For example, the operands to the Cyber ALU operations must be registers; all data accesses occur during loads and stores. If the actual machine is also a load/store architecture, e.g. IBM's 801 [11] and UC Berkeley's RISC [10], the distortion would be slight. However designs similar to DEC's VAX that heavily utilize memory operands would show a higher data fetch to instruction ratio. We expect the simulation results to be adequate for high-level design decisions; low-level design decisions can only be made by simulating the actual machine.

4. The NYU Ultracomputer

The NYU Ultracomputer is fully described in [5]; a brief overview is given here. The Ultracomputer is a hardware approximation to the abstract (physically unrealizable) "paracomputer" model of computation. In this model each processor may read a (local or global) memory location in a single cycle, even if multiple PEs simultaneously access the same location. Simultaneous writes are also accomplished in one cycle. The target memory location will contain the data value written by some PE just as if the writes had occurred in some unspecified serial order. This atomicity of concurrent requests is called the serialization principle.

Because of this simple approach to memory access conflicts, the clocking scheme for the simulation can be very simple. It is further assumed that all instructions execute in one clock cycle (including any memory references generated). In fact, Washcloth uses a single global clock, which is advanced by one after every PE has executed one instruction.

Synchronization of the PEs is accomplished via a special instruction, fetch-and-add. The format of this operation is FetchAdd(V,e), where V is an integer variable and e is an integer expression. This indivisible operation yields the sum S=V+e as its value and replaces the contents of storage location V by this sum S. Moreover, FetchAdd satisfies the serialization principle. A more complete description of the fetch-and-add operation can be found in [5] and [6].

The fetch-and-add primitive can be used to implement a truly parallel queue; that is, (many) queue insertion and queue removal operations can be executed concurrently and will complete in about the time required for just one operation. This enables the operating system to support a style of programming where programs are split up into tasks. As tasks are created, they are inserted into a ready queue. Whenever a PE finishes a task, it removes the next task from the ready queue. Our experience has shown that parallel programs are fairly easy to write and debug using this style.

Figure 1 contains a very simple, sample, serial program that performs a numerical integration of the function F on the interval [0..1] by subdividing the interval and estimating the area of each subinterval by a rectangle. In Figure 2 this is converted into a sample Ultracomputer program written for the Washcloth simulator. Each PE is assigned a set of contiguous subintervals to calculate; the partial results from each processor are stored into a global memory vector, and the last processor to finish calculating its subintervals then reduces this vector to produce the final result.

5. The UI Cedar machine

The UI Cedar Machine is described in [3]; a brief overview is given here. Although the machine design groups PEs into clusters, the simulator design does not yet support this concept. Cedar allows only one PE to access a memory location (or more accurately a memory module) in any given cycle. Attempts at simultaneous access by several PEs will result in one request for a single (unspecified) PE being serviced in each cycle. The remaining PEs must wait for subsequent cycles when the memory module is free.

Because of this different approach to memory access conflicts, the clocking scheme for the simulation must be more complicated than that used for the Ultracomputer. For Cedar, a separate clock is kept for each individual PE. A single global clock is advanced cycle by cycle; at each point, all PEs are checked to see if their individual clock is less that the global clock. If so, the PE is allowed to execute an instruction. Each global memory reference locks the target memory location; any attempt to access a locked location is delayed.

In order to simulate a high-level data flow machine such as Cedar, several additional mechanisms are required. These primarily relate to defining and implementing the control flow required in a data flow machine. In essence, the data flow must be implemented within the Cyber's Von Neumann architecture. Also, various synchronization primitives not present in the Ultracomputer must be defined.

Synchronization in Cedar is achieved in three ways. The first is by virtue of the high-level data flow analysis provided by the Cedar Fortran compiler. The compiler restructures the program into "compound functions" (CPFs) and "control functions" (CTFs). Collectively, these items can be considered to be the nodes of a standard data flow graph. The Global Control Unit (GCU) executes the CTFs; the CPFs are given to one or more PEs to execute. Whenever a CPF or CTF is finished, the GCU determines which, if any, of the nodes successors can be executed. All such nodes are scheduled for execution. It should be noted that the GCU may either be a specialized piece of hardware or be a component of the operating system running on each PE. It is expected that both schemes will be simulated.

The second method of synchronization in Cedar is for use by the several PEs that may be assigned to execute a particular CPF. This is accomplished by synchronization primitives, which are indivisible read-modify-write type operations. Whereas the Ultracomputer has only one such operation, fetch-and-add, the Cedar instruction set allows many operations to be executed in a read-modify-write fashion. Each of these operations has its own opcode in the Washcloth simulator.

The third method of synchronization in Cedar is also for use by the several PEs that may be assigned to execute a particular CPF. All global memory locations have a full/empty bit associated with them. Special instructions exist to set and test these bits.

The Cedar philosophy for producing programs is quite different from that of the Ultracomputer. Instead of restructuring programs for parallel execution by traditional hand programming techniques, Cedar plans to provide several compiler tools to accomplish this. A typical programming effort would entail the user feeding a standard serial program into the Paraphrase analyzer [7],[8]. Next, the Paraphrase output would be given to a scheduling program called EWE [1]. The output of EWE will be in "Cedar Fortran", which can compiled into code for the Cedar machine. The user is able to make suggestions and changes by hand, and, if necessary, Paraphrase or EWE can be rerun. Since we do not have a Cedar Fortran compiler, a translator will convert the output of EWE into regular Fortran (with the proper extensions). Figure 3 shows a simple, sample Cedar program before being converted to Washcloth format. Note that EWE uses the comments starting with the word "CEDAR" to describe the data flow graph and other non-Fortran constructs; by happy coincidence, the word "CEDAR" starts with a "C" and is therefore a Fortran comment.

6. Summary

In summary, the Washcloth simulator has proven to be a flexible and powerful tool for computer architecture research. It has been used to simulate several parallel processor designs and with minor modifications, Washcloth should continue to provide fast, efficient simulations of new parallel computer architectures.

```
      subroutine INTEGRA
      common numint,sum

      sum = 0.0
      do 10 i = 1,numint
10      sum = sum + F(  (float(i) - 0.5) / float(numint)  )

      sum = sum / float(numint)
      end

      real function F(x)
c
c  Trivial version-- f(x) = x
c
      F = x
      end
```

Figure 1. Serial version of integration program

```
      subroutine INTEGRA
c
c  Global storage
c
c numpe   is the number of PEs to use
c N       is the maximum number of PEs that may be assigned
c The first common line contains items necessary for Washcloth operation
c
      common avail(600),botpub,lowert,uppert
      common numpe,numint,donepes,sum,sumvec(0:N)
      integer donepes
c
c  All other data is local
c
      integer penum,frstint,lastint
      real lsum
c-----------
c  The following code is executed by PE0, PE1, PE2,... PE(numpe-1)
c
c Determine the subintervals assigned to this PE
c
      penum   = irpn(dummy)            ; Read the PE number
      frstint = penum * (numint / numpe) + 1
      lastint = (penum + 1) * (numint / numpe)
      lsum    = 0.0
c
      do 10 i = frstint, lastint
10      lsum  = lsum + f(  (float(i)-.5) / float(numint)  )
c
      sumvec(penum) = lsum
c
c  All PEs except the last one go into an infinite loop (inloop).
c
      if ( irepad(donepes,1) .lt. numpe ) call inloop
c-----------
c The following code is executed by the one PE which finishes last
c
      lsum = 0.0
      do 20 i = 0, (numpe - 1)
20      lsum = lsum + sumvec(i)
      sum  = lsum / float(numint)
      end
```

Figure 2. A sample Ultracomputer program for Washcloth

```
CEDAR GCU 0 0 SUBROUTINE (1)
        subroutine INTEGRA
CEDAR MALLOC PRIVATE
        real lsum
        integer frstint,lastint
CEDAR MALLOC FULLY-SHARED
c N is the number of PEs assigned to CPF 1
        real sum,sumvec(N),sum
        integer numint

CEDAR GCU 1 1 CPF N (2)
        doall penum = 1, N

        frstint = penum * (numint / numpe) + 1
        lastint = (penum + 1) * (numint / numpe)
        lsum    = 0.0
c
        do 10 i = frstint, lastint
10        lsum  = lsum + f(  (float(i)-.5) / float(numint)  )
c
        sumvec(penum) = lsum

CEDAR GCU 2 1 COMPUTE (3)
c
        lsum = 0.0
        do 20 i = 1, N
20        lsum = lsum + sumvec(i)
        sum  = lsum / float(numint)

CEDAR GCU 3 1 RETURN
```

Figure 3. A sample Cedar program ready for input to the preprocessor

REFERENCES

(1) Ron Cytron, Harlan Husmann, and Sam Midkiff, "Cedar Fortran: A Language for EWE", Cedar note #33, Laboratory for Advanced Supercomputers, University of Illinois, Champaign, Illinois, Feb. 1984.

(2) E. Draughton, R. Grishman, J. Schwartz, and A. Stein, "Programming Considerations for Parallel Computers", IMM 362, Courant Institute of Mathematical Sciences, New York University, New York. Department of Computer Science.

(3) Daniel Gajski, David Kuck, Duncan Lawrie, and Ahmed Sameh, "CEDAR -- A Large Scale Multiprocessor", *Intl. Conf. on Parallel Processing,* Aug. 1983, pp. 524-529.

(4) Allan Gottlieb, "WASHCLOTH -- The Logical Successor to Soapsuds", Ultracomputer Note #12, Courant Institute of Mathematical Sciences, New York University, New York, Department of Computer Science, Dec. 1980.

(5) Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolf, and Marc Snir, "The NYU Ultracomputer -- Designing an MIMD Parallel Machine", *IEEE Transactions on Computers,* Vol. C-32, No. 2, Feb. 1983, pp. 175-189.

(6) Allan Gottlieb, and Clyde P. Kruskal, "Coordinating parallel processors: A partial unification", *Comput. Arch. News,* Oct. 1981, pp. 16-24.

(7) David J. Kuck, R.H. Kuhn, B. Leasure, and M. Wolfe, "The Structure of an Advanced Vectorizer for Pipelined Processor", *Proceedings of COMPSAC 80, the 4th International Computer Software and Applications Conference,* Chicago, Illinois, pp. 709-715, Oct. 1980.

(8) David J. Kuck, R.H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependence Graphs and Compiler Optimizations", *Proceedings of the 8th ACM Symposium on Principle of Programming Languages,* Williamsburg, Virginia, PP. 207-218, Jan. 1981.

(9) Kevin P. McAuliffe, Ph.D. thesis Courant Institute of Mathematical Sciences, New York University, New York, Department of Computer Science, in preparation.

(10) D. A. Patterson, and C. H. Sequin, "RISC I: A Reduced Instruction Set VLSI Computer", *Proc. Eighth Intl. Symposium on Computer Architecture,* May 1981, pp. 443-457.

(11) G. Radin, "The 801 Minicomputer", *Proc. Symposium on Architectural Support for Programming Languages and Operating Systems,* Mar. 1982.

(12) Marc Snir, "NETSIM -- A Network Simulator for the Ultracomputer", Ultracomputer Note #28, Courant Institute of Mathematical Sciences, New York University, New York, Department of Computer Science, 1981.